# Protecting algorithms from frameworks

V. Balaji, Peter Phillipps, Alex Pletzer, Will Cooke, Alistair Adcroft

Princeton University and NOAA/GFDL

FMS Developers’ Forum

Princeton NJ

8 March 2005

## Programming styles

... have evolved from simple ***formula translation*** and direct addressing of variables to something more abstract.

- “Object-oriented programming”.
- “Component-based design”.
- “High-level standards based on abstract datatypes”.

## Performance crisis

The ***abstraction penalty*** is high. Much of the crisis revolves around the f90 array model:

- array syntax.
- assumed-shape arrays degrade data flow.
- The **`pointer`** attribute introduces "aliasing" problems.

The tradeoffs:

- conformance checking
- flexibility in stride and layout
- readability

  ***versus*** ...

- performance.

## Key quotes

- “Do not yield to CSBS.”
- “Have we been sold a bill of goods?”
- “At a sufficient level of abstraction, the universe is mostly empty space.”

## Array arguments: “f77” versus “f90”

- f90:

```
real ::  a(ni,nj,nk)
...
call sub(a)
subroutine sub(a)
real, intent(in) ::  a(:,:,:)
```

(1)

- f77:

```
real ::  a(ni,nj,nk)
...
call sub(a,ni,nj,nk)
subroutine sub(a,ni,nj,nk)
integer, intent(in) ::  ni, nj, nk
real, intent(in) ::  a(ni,nj,nk)
```

(2)

The f77 style calling syntax has a higher probability ($\sim$1) of passing the array by reference, and the f90 style has a higher probability of generating an array temporary.

## Data copies

Compilers have shown themselves unable to solve the problem of deciding when an array argument is ***well-formed*** (contiguous, unit stride). When they find this ambiguity, they generate an array temporary, and a memory copy, for safety.

A simple test case based on MOM4 shows that the behaviour is different depending on whether

- array is held within a type (**`foo%bar(:,:,:)`**)
- array has **`pointer`** or **`allocatable`** attribute
- array is inherited from module or passed as argument
- array uses static or dynamic allocation.

We have requested SGI and Intel compiler groups over the years to provide pragmas or compiler directives (**`!dir$`**, **`#pragma`**) whereby the user can **force** the generation of an array temporary at one spot, and **prohibit** it in another.

## Portability

None of this behaviour is predictable, and none of the experience carries over from one platform to the next.

- Altix (and many other platforms) have implemented the Fortran 2000 extension of allowing `allocatable` arrays within types. We have shown on Altix that this significantly improves performance of certain data structures in FMS.

```
#ifdef __ia64
#define _ALLOCATABLE allocatable
#define _ALLOCATED allocated
#else
#define _ALLOCATABLE pointer
#define _ALLOCATED associated
#endif
type ::  foo
real, _ALLOCATABLE ::  bar(:)
end type
...
if( .NOT. _ALLOCATED(foo%bar) )allocate( foo%bar(n) )
```
(3)

- On IBM, dynamic allocation outperforms static!

## Frameworks

Software frameworks promise to deliver a parallel programming model that:

- is high performance
- is easy to use
- uses high-level parallel programming constructs to hide underlying architectural complexity
- provides portable, interchangeable software modules that can be exchanged across the entire community.

The major disadvantage is that frameworks tend to be ***pervasive***. For instance, it is now not easy to detach a physics module from FMS and make it usable within some other model... almost every FMS module has a dependency on our own error handler, diagnostics manager, restart I/O and MPP.

Community-wide standards (ESMF, PRISM) are still very immature. Their pervasive nature means a significant and risky long-term commitment.

## Protecting algorithms from frameworks

The dilemma is that models and platforms are both now too complex to tolerate the more direct programming style where the user retains control over memory management, and over data flow. Yet, the more abstract programming style that is needed in the current software environment has significant performance shortfalls, and considerable sunk costs for framework adoption.

Is there a way to develop algorithms where the data flow is more controlled, yet orchestrated at high level using abstract logic and datatypes?

We propose a programming model which we call the **kernel-driver** programming model.

## The kernel-driver programming model

- Software is organized into **modules** that perform a certain operation. The granularity of modules is still to be decided: a module could be a dynamical core (FV), a physics package (radiation, cumulus momentum transport), a diffusion scheme, or a differential operator.

- The module has **state** variables where an input state is passed in, and the modified state or tendency is passed out. There may be multiple instances of a state within the same run.

- The **driver** is a high-level ***public*** routine, typically working with abstract datatypes expressing the state, and allied with some software **framework**. The driver performs most ***technical*** functions associated with program execution, such as managing parallel data dependencies, I/O, and exception handling. Usually these functions are associated with a framework. Drivers associated with multiple frameworks (FMS, ESMF, ...) may be provided.

- The driver calls a **kernel** for performing the algorithmic function of the module. The kernel is a pure ***serial*** routine operating entirely through intrinsic types and argument lists. The interface is designed to pass all arrays by reference. There may be multiple kernels associated with a driver (e.g B- and C-grid versions of a dynamical routine, different flavours of a scheme). The driver may have to call the kernel several times in a single pass to complete updating a domain (examples below). The kernel is a ***private*** routine; only the driver can call it. But it can be private to different modules, with different drivers.

## Driving forces at GFDL

- FMS framework adoption is proving to be a significant barrier for some of the more recent added models: HIMF and ZETAC. ESMF adoption possibly has an even higher barrier.

- For ZETAC in particular, we saw the need for redesign of the AM physics calling tree to support the particularities of the NH core (regionality, high-resolution). We used this as an excuse to study the AM physics interfaces again from a design and dataflow standpoint, to see if we could arrive at something of benefit to everyone in terms of increased flexibility.

- Hybrid ocean core development over the next few years requires the ability to develop and do extensive road tests on algorithms in a framework-neutral manner. This discussion has been informally called the “GOLLUM project”.

## Refactoring guidelines

At this point, we learnt that there’s a technical name, a field, probably some journals, and no doubt some world experts with tenure, on what we’re doing... **refactoring!**

**`http://www.linuxjournal.com/article/8087`**

The AM physics redesign got underway with the following guidelines:

- no answer changes, and
- no performance degradation!

## Structure of module `foo`

- **`module foo_mod`**

  defines all the public interfaces of **`foo`**. This may include a public datatype **`foo_type`** to hold the state variable, which may contain many scalar parameters and many state arrays.

- **`subroutine foo_init`**

  called once to initialize the module.

- **`subroutine foo_exit`** or **`foo_end`**

  called once to terminate the module.

- Unit test program.

```
#ifdef test_foo
program foo_unit_test
...
#endif
```

(4)

## Structure of module `foo`: managing instances of `foo_type`

- `subroutine foo_new(foo_inst)`

  `type(foo_type), intent(out) ::  foo_inst`

  creates a new instance of `foo_type`. Registers diag fields, reads input parameters, initializes data arrays.

- `subroutine foo_del(foo_inst)`

  `type(foo_type), intent(inout) ::  foo_inst`

  destroys an instance of `foo_type`.

## Structure of module `foo`: driver and kernel

- `subroutine foo(foo_inst,...)`

  `type(foo_type), intent(inout) ::  foo_inst`

  The driver routine may update the input itself, or fill or increment another type holding a tendency. The driver calls diagnostics routines, halo updates, error handling routines.

  Sets up arrays to call the kernel, and performs all correctness and conformance checks. Performs one or a series of calls to the kernel.

- `subroutine foo_kernel(bar,i,...,rc)`

  ```
  integer ::  i(18)
  real, intent(inout) ::  bar(i(1):i(2),i(3):i(4)...)
  ```

  ```
  call foo_kernel( foo%bar(isd,jsd,1), (/is,ie,js,je,.../), ...  )
  ```

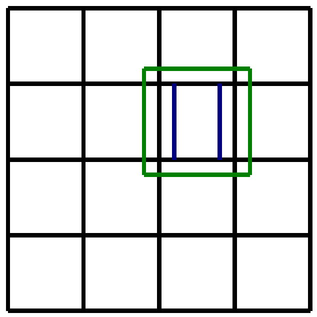
  `foo_kernel` is a private routine that has no dependencies on any framework, and typically is called through a framework-related driver. In case of error, it does not stop, but returns an error code `rc`, and the driver must take appropriate action.

  Since the kernel has no dependencies, it can be archived and called directly as an external library routine. Of course, in this case, all the safety features and technical functions usually supplied by a framework are absent.

## The kernel interface: the index array

The kernel is a serial routine called by a parallel framework. The input index array tells the kernel how to dimension the array, what portion of the domain has valid data, and what subset of the domain to operate upon.

Consider an example of a parallel code where the algorithm has a dependency on **(i+1,j)**. One possible way to overlap communication with computation is to have the driver make 3 separate calls to the kernel, from left to right on the regions shown with blue lines, interspersed with non-blocking halo update calls.



```
call foo_kernel(1)
call mpp_update_domains(...)
call foo_kernel(2)
call mpp_update_domains(...,COMPLETE)
call foo_kernel(3)
```

(5)

Alternatively, the same kernel could be called from a cache-blocked model, with arrays structured as **(i,j,k,n)**, where the block loop over **n** is placed in the driver. A single 3D kernel can serve the needs of codes using either 3D or 4D cache-blocked array patterns.

## The kernel interface: other arguments

- Should physical constants also be passed through an argument list?
- Should we permit optional work arrays?
- Should we permit automatic arrays within the kernel?
- Should the code be entirely thread-safe?
- Should its entire state be contained in its arguments?

Many such questions cannot be answered from first principles, but by prototyping actual code. A lot depends on the appropriate granularity at which we designate something to be a module, with its own kernel. We are still working through these issues.

Let us now look at some code examples.

## Problems with the `pointer` attribute

- Compiler cannot assume pointer is static.

```
real, pointer ::  a(:)
call sub(a)
subroutine sub(a)
real, intent(inout), pointer ::  a(:)
```
(6)

A memory copy of **a** is generated.

- Pipeline optimizations are inhibited.

```
real, pointer ::  a(:)
a(:)  = b(:)
```
(7)

Compiler has to allow for the possibility that **a** and **b** have overlapping memory locations: **aliasing.**

## Problems with `pointer` attribute: memory leaks

Two ways to initialize a pointer array:

- Association:

```
real, pointer ::  a(:)
real ::  b(n)
a => b
...
nullify(a)
```
(8)

- Allocation:

```
real, pointer ::  a(:)
allocate( a(n) )
...
nullify(a)
```
(9)

which actually does the following:

```
real, pointer ::  a(:)
real ::  _tmp(:)
allocate( _tmp(n) )
a => _tmp
...
nullify(a)
```
(10)

In the case of CodeBlock 10, the nullifi cation leaves a dangling array with no handle!

A safe way is to use **`deallocate(a,stat=rc)`**, which will never abort. The safer way is to use the f2k extension where possible and avoid pointers. A MOM4 test at OM3 resolution shows a 20% improvement on the Altix.